Error-Correcting Graphs for Software Watermarking

Christian Collberg[1*], Stephen Kobourov[1**], Edward Carter[1*], and Clark Thomborson[2***]

[1] Department of Computer Science, University of Arizona
{collberg,kobourov,ecarter}@cs.arizona.edu
[2] Department of Computer Science, University of Auckland
cthombor@cs.auckland.ac.nz

Abstract. In this paper, we discuss graph-theoretic approaches to software watermarking and fingerprinting. Software watermarking is used to discourage intellectual property theft and software fingerprinting is used to trace intellectual property copyright violations. We focus on two algorithms that encode information in software through the use of graph structures. We then consider the different attack models intended to disable the watermark while not affecting the correctness or performance of the program. Finally, we present several classes of graphs that can be used for watermarking and fingerprinting and analyze their properties (resiliency, data rate, performance, and stealthiness).

1 Introduction

Watermarking and fingerprinting [2,3] are popular techniques for protecting the intellectual property of digital artifacts such as images, audio and video. A watermark is a copyright notice that is embedded into the artifact that uniquely identifies its owner. A fingerprint is similar but identifies the *customer* who purchased the copy. Watermarking an object discourages intellectual property theft, whereas fingerprinting allows us to trace the copyright violator.

Several algorithms for watermarking *software* have recently been developed [4,7,13,18]. We focus on two of these algorithms: one by Collberg and Thomborson [4] and the other by Venkatesan *et al.* [18]. Both of these algorithms encode the watermark using graphs which are then embedded into the cover program.

In the remainder of this section we formally present the software watermarking and graph encoding problems. In Section 2 we consider the algorithms of Collberg and Thomborson [4] and by Venkatesan *et al.* [18] in detail. In Section 3 we present models of how an adversary can attack a program to destroy the watermark (Section 3), and present classes of graphs suitable for encoding watermarks (Section 4). In Section 5 we evaluate the performance of the new encoding scheme that uses reducible permutation graphs. In Section 6 we summarize our results and consider future work.

1.1 Software Watermarking and Fingerprinting

While watermarking and fingerprinting have different uses, they both rely on encoding a number inside a program. For convenience, in the rest of this paper we refer to both processes as watermarking. Similarly, we assume that the watermark is an integer W. A *software watermarking system* can be formally described by the following three functions:

$$\mathcal{E}(P, W, k) \rightarrow P_w \qquad \mathcal{R}(P_w, k) \rightarrow W \qquad \mathcal{A}(P) \rightarrow P'$$

The *embedder* $\mathcal{E}$ embeds the watermark W into the (cover) program P using the secret key k, yielding a watermarked program P_w. The *recognizer* $\mathcal{R}$ extracts W from P_w, also using k as the key. The *attack function* $\mathcal{A}$ models the types of attacks that an adversary may launch against a watermarked program in order to foil recognition. For example, a common attack model includes semantics-preserving transformations such as code optimization and binary translation. Attacks can also be non-semantics-preserving, allowing the attacker to alter the meaning of the program.

A good watermarking system must have the following properties:

* Partially supported by the NSF under grant CCR-0073483 and by the AFRL under contract F33615-02-C-1146.
** Partially supported by the NSF under grant ACR-0222920.
*** Partial upport by New Economy Research Fund of New Zealand under contracts UOAX9906, UOAX0214.

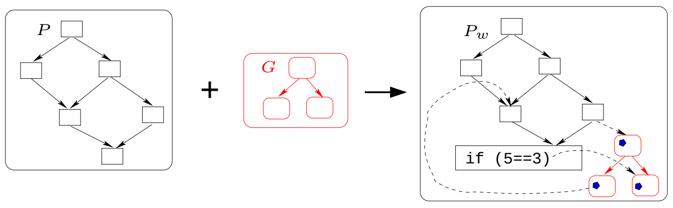


Fig. 1. The GTW algorithm. Dashed control-flow edges connect the watermark graph G with the cover control-flow graph P The watermark nodes of P_w are *marked* to distinguish them from the nodes of P.

high resiliency: Most importantly, a watermarking system must be *resilient* against a reasonable set of de-watermarking attacks. In this context, resiliency refers to the ability to recognize a watermark even after the watermarked program has been subjected to an attack, i.e., $\mathcal{R}(\mathcal{A}(\mathcal{E}(P, W, k)), k) \rightarrow W$;
high data rate: The ratio of the number of bits encoded by the watermark W to the total size of the watermark should be high;
high stealth: To prevent an adversary from easily locating the watermark, P and P_w should have similar statistical properties;
high performance: The watermarking should not adversely affect the size and execution time of P_w.

Note that any software watermarking technique will exhibit a trade-off between resilience, data rate, performance, and stealth. For example, the resilience of a watermark can easily be increased by exploiting redundancy (i.e., including the mark several times in the cover program), but this will lead to reduced data rate. Similarly, a watermark such as ⌜`static int watermark=314`⌝ has high performance but low stealth.

1.2 Graph Encodings

In this paper we focus on two software watermarking algorithms that encode watermarks as graph structures. In general, such encodings make use of an encoding function e that converts the watermark W into a graph G, $e(W) \rightarrow G$, and also of a decoding function d that converts a graph into an integer, $d(G) \rightarrow W$. We call the pair (e, d) a *graph codec*. From a graph-theoretic point of view, we are looking for a class of graphs $\mathbb{G}$ and a corresponding codec $(e, d)_{\mathbb{G}}$ with the following properties:

appropriate graph types: Graphs in $\mathbb{G}$ should be directed multigraphs with an ordering on the outgoing edges. Furthermore, graphs in $\mathbb{G}$ should have low max out-degree, to match graphs from real programs;
high resiliency: The decoder function $d(G)$ should be insensitive to small perturbations of G (the result of adversarial attacks against the watermarked program) such as insertions or deletions of a constant number of nodes and/or edges. Formally: $G \in \mathbb{G}, d(G) \rightarrow W, G' \approx G \Rightarrow d(G') \rightarrow W$;
small size: The size $|P_w| - |P|$ of the embedded watermark should be small;
efficient codecs: $(e, d)_{\mathbb{G}}$ should be polynomial time efficient.

2 Graph-Based Watermarking Algorithms

Software watermarking algorithms fall in two broad categories, *static* and *dynamic*. A static algorithm recognizes the watermark by examining the (source or compiled) code of the watermarked program. A dynamic algorithm recognizes the watermark by examining the state of the program after it has been executed with a special finite input sequence $i_1, i_2, \ldots, i_n$. Thus, for a dynamic algorithm the recognizer will have the signature $\mathcal{R}(P_w [\![i_1, i_2, \ldots, i_n]\!]) \rightarrow W$, where $P [\![I]\!]$ is the state of program P after input I.

2.1 The GTW Algorithm

Static watermarking algorithms typically embed the watermark: by permuting segments of code, such as statements, basic blocks, or arms of switch-statements [7], or by altering instruction frequencies [15], or by inserting a code segment which has no semantic effect. The Graph-Theoretic Watermark (GTW) algorithm of Venkatesan *et al* [18] falls in the last category. The basic embedding technique is as follows:

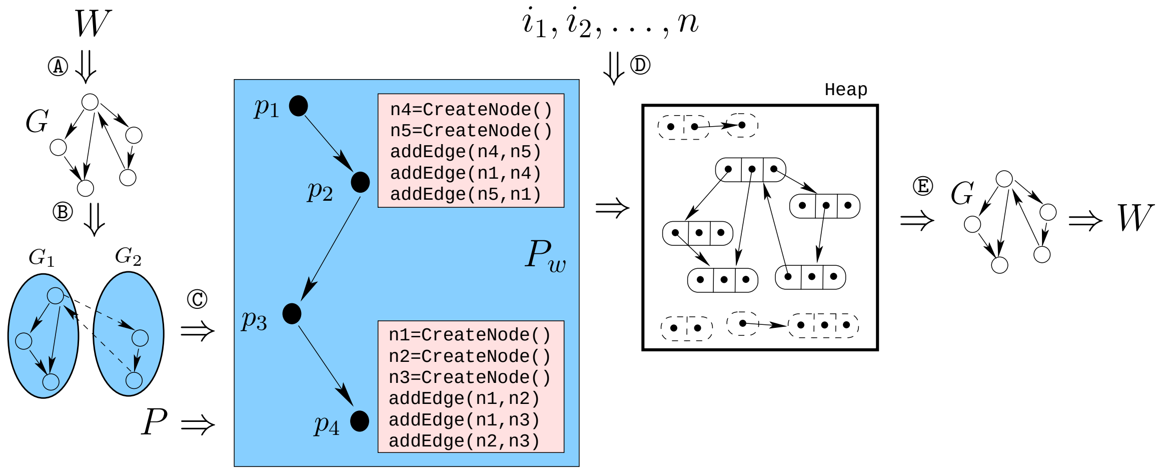


Fig. 2. The Collberg-Thomborson watermarking algorithm.

1. encode the integer watermark value W as a control-flow graph (CFG) $G \leftarrow e(W)$,
2. construct the watermarked program P_w by combining G and the program CFG P,
3. connect G and P by adding bogus control-flow edges,
4. mark the nodes (the *basic blocks*) of G.

This process is illustrated in Figure 1. In order for the watermark to be stealthy G and P should be tightly integrated. In [18], integration is achieved by performing a random walk over the nodes of G and P and adding edges until P_w is uniformly dense. The watermark is extracted from P_w as follows:

1. make the nodes of the watermark graph G be the set of marked blocks from P_w,
2. make the edges of G be the set of control-flow edges between marked blocks in P_w,
3. compute the watermark using the decoder $d(G)$.

Once again, in order for the watermark to be stealthy, the watermark graph G should look as much like "real" control-flow graphs as possible. Thus, the following properties must hold:

– G should be *reducible* [1]. Reducible graphs are constructed from structured programs, using only properly nested statements such as if-then-else, while-do, repeat-until, etc. Well-structured CFGs can be approximated by series-parallel graphs [10].
– G should have low maximum out-degree. In real code, basic blocks have out-degree 1 or 2. Only switch-statements construct CFGs with out-degree greater than 2, and these are unusual in most programs.
– G should be "skinny". In typical functions, control structures (loops and conditionals) are nested only a few levels deep. Hence, the width of G should be a small constant.

2.2 The CT Algorithm

The algorithm of Collberg and Thomborson [4] (henceforth, *CT*) embeds the watermark in a dynamically built, linked, graph structure. The rationale for this design is that graph-building code is difficult for an adversary to analyze, due to aliasing effects [8]. (The *aliasing problem* determines whether two pointers may/must refer to the same memory location. In the general case this is known to be undecidable [14].) The algorithm is illustrated in Figure 2:

1. At Ⓐ the watermark number is encoded into a graph G.
2. At Ⓑ G is split into a number of subgraphs.
3. At Ⓒ each subgraph is converted into a code sequence that builds the graph. P_w is constructed by inserting the graph-building statements along a special execution path $\langle p_1, p_2, p_3, p_4 \rangle$ of the program.

4. At Ⓓ, the recognition sequence is started by executing P_w with a special input sequence $i_1, i_2, \ldots, i_n$. This causes the program to follow the execution path $\langle p_1, p_2, p_3, p_4 \rangle$ and the watermark graph G to be built on the *heap.* (A *heap*, in this context, is the memory in a running program where dynamically allocated objects are allocated.)
5. At Ⓔ, G is extracted from the heap and decoded into W.

In order for the watermark to be stealthy, the CT algorithm requires that the watermark graph G look like a "real" dynamic data structure. Thus, the following properties must hold:

– G should have a low maximum out-degree, typically 2 or 3. Most linked data structures used in real programs are linked lists, binary trees, and sparse graphs, which all have low constant out-degree.
– G should be have a unique root node from which every other node is reachable. Data-structures in real programs are usually passed around using such root nodes.

3 Models of Attack

A successful attack against P_w prevents the recognizer from extracting the watermark while not seriously harming the performance or correctness of the program. It is generally assumed that an attacker has access to the algorithms used by the embedder and recognizer. Only the watermark key is kept hidden. We distinguish between *semantics-preserving* and *non-semantics-preserving* attacks. A semantics-preserving attack is typically *automatic* and consists of running a set of semantics-preserving transformations over P_w. The transformations may reorganize the code, optimize the code, insert bogus code, remove or add abstractions, etc. in order to confuse the watermark recognizer [5,6]. The advantage of this type of attack is that the attacker does not need to know where in P_w the watermark is hidden. Rather, he can repeatedly apply semantics-preserving transformations uniformly over P_w until the code is sufficiently convoluted to confuse the recognizer.[3] A non-semantics-preserving attack requires the attacker to "guess" the approximate location of the watermark code. This can either be done through manual examination or through statistical analysis. The attacker will then proceed to carefully make minor "tweaks" to the code in an attempt to disable the watermark while not disrupting the normal execution of the program. In the remainder of this section we discuss attacks against graph-based software watermarking algorithms.

3.1 Edge-Flip Attacks

An *edge-flip attack* reorders the outgoing edges of each node in the graph. In the GTW algorithm this amounts to transforming $\ulcorner$`if (`p`)` T `else` $E\urcorner$ into $\ulcorner$`if (`$!p$`)` E `else` $T\urcorner$. In the CT algorithm this attack amounts to reordering the fields within the graph node structure and making appropriate modifications to any code that references the structure. For both algorithms these are simple semantics-preserving attacks that have little or no performance overhead.

3.2 Edge-Addition/Deletion Attacks

In the GTW algorithm a graph edge corresponds to a possible control-flow in the program. Adding an edge is accomplished using opaque predicates. For example, to add an edge between basic blocks B_1 and B_2 an attacker can add a bogus jump from B_1: $\ulcorner$`if (`P^F`) goto` $B_2\urcorner$. P^F is an *opaque false predicate*, a boolean expression that always evaluates to false but which is difficult for an adversary to evaluate. This is a simple attack with low performance overhead, assuming the opaque predicate is cheap [6].

In the CT algorithm a graph edge is a link between two heap objects. Adding an extra edge requires the attacker to extend each graph node with a bogus pointer field and adding code to link nodes together on this field. This can be an extremely dangerous operation since it may affect which nodes get garbage collected and hence may introduce a memory leak into the program.

For both algorithms edge-deletions are dangerous operations for the attacker. In the GTW algorithm it would require the attacker to remove a branch from the program. In the CT algorithm a link would have to be removed from a data-structure.

[3] We assume that it is possible for the attacker to study the recognition algorithm to determine the kinds of semantics-preserving transformations that may be effective against a particular watermarking system.

3.3 Node-Addition/Deletion Attacks

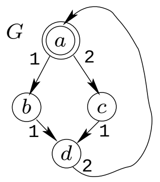
Adding a node in the GTW algorithm can be easily accomplished using opaque predicates. It is simply a matter of inserting the code ⌜`if` (P^F) `B`⌝ into the code, where B is a bogus basic block. Node additions are also easy in the CT algorithm, and similar to adding edges. Node deletions are difficult for both algorithms since they require a deep understanding of the semantics of the underlying program.

3.4 The Attack Model

Based on these observations, we can formulate a model for attacks against graph-based software watermarks. From what we have seen, not all attacks will be effective against all watermarking systems.

Graphs are multi-graphs represented as a pair $(root, edges)$ where *edges* is list of triples $from \xrightarrow{link} to$, where *link* is the number (label) of the outgoing edge. For every node a we require that there be no outgoing edges with the same link number.

For example, the graph G on the left would be represented by the tuple on the right:



$$G = (a, \langle a \xrightarrow{1} b, a \xrightarrow{2} c, b \xrightarrow{1} d, c \xrightarrow{1} d, d \xrightarrow{2} a\rangle)$$

An attack against a watermark graph G is modeled by transferring G over a lossy communications channel C. The severity of the attack is modeled by the lossy properties of C. A edge-flip attack only affecting a's outgoing edges would look like this:

$$(a, \langle a \xrightarrow{1} b, a \xrightarrow{2} c, b \xrightarrow{1} d, c \xrightarrow{1} d, d \xrightarrow{2} a\rangle) \overset{C}{\Longrightarrow} (a, \langle a \xrightarrow{2} b, a \xrightarrow{1} c, b \xrightarrow{1} d, c \xrightarrow{1} d, d \xrightarrow{2} a\rangle)$$

Given a definition of a lossy channel C our goal is to construct a class of graphs $\mathbb{G}$ that can be transfered across C unscathed. We call such graphs *error-correcting graphs*. For example, in the next section we will present *reducible permutation graphs* which are resilient to edge-flip attacks.

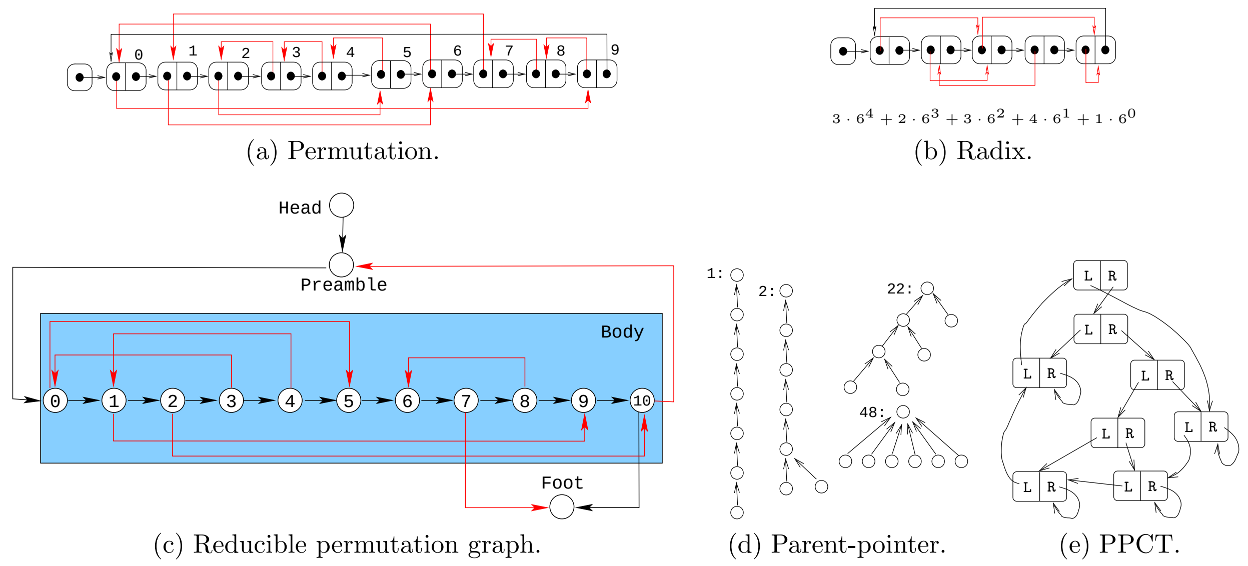
Other types of error-correcting graph models can be considered. For example, it would seem natural to use an adjacency matrix representation of the graph, encode this as a bitstring, and use error-correcting codes [17] to protect it as it is being transfered across the lossy channel. However, for the purposes of software watermarking this model is not as good as the lossy channel model. For example, both the GTW and CT algorithms use graphs that are inherently linked structures. GTW's control-flow graph is implicit in the control-flow of the program and cannot be represented any other way. The CT algorithm gets its strength from the fact that it is a linked data-structure which is known to be computationally difficult to analyze. An adjacency matrix representation would not have this property.

4 Graph Encodings

Any enumerable class of graphs $\mathbb{G}$ may be used for software watermarking, giving us the freedom to choose a class with particularly desirable properties. To obtain a watermark with efficient encoding and decoding functions, we require the enumeration to be polynomial-time efficient. Stealthy watermarks are chosen from a class of graphs, all of whose members resemble a naturally-arising population of graphs. Watermarks with high data rate are chosen with a uniform probability distribution from a very large class of graphs, each member of which can be embedded with a relatively small amount of overhead in space.

To lower-bound the data rate of a watermarking scheme, we assume that each graph g in $\mathbb{G}$ is chosen as a watermark with equal likelihood, and that the extra space required to embed any such g as a watermark in our target program is no more than s bytes. The data rate of this watermarking scheme is then lower-bounded by $\lg |\mathbb{G}|/s$, and we call this a "high data rate" if it is at least one bit of watermarking information per word of overhead space.

We have been particularly interested in $\mathbb{G}$ that resemble either the linked-list data structures or the trees that are built dynamically by many "naturally occurring" programs.


0
1
2
3
4
5
6
7
8
9
$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0$
Head
Preamble
Body
0
1
2
3
4
5
6
7
8
9
10
Foot
1:
2:
22:
48:
L
R

(a) Permutation. (b) Radix.

(c) Reducible permutation graph. (d) Parent-pointer. (e) PPCT.

Fig. 3. Graph encodings.

4.1 Parent-Pointer Trees

The parent-pointer data structure is the most space-efficient representation of a tree using nodes with pointer fields. Each node in the data structure has just one pointer field referencing its parent. We may take our class of watermarking graphs to be $\mathbb{G}_{pp}$, the set of all parent-pointer graphs on n nodes, for some suitable constant n; see Fig. 3(d). This class is stealthy, for structures with a single pointer field arise often in programs that would be watermarked. If we take $n = 655$ nodes, then $|\mathbb{G}_{pp}| \approx 2^{1024}$ by the enumeration described in Knuth Vol. 1 [11]. Since one word of storage is required for each pointer in our watermark, the data rate is high: $1024/655 = 1.56$ bits per word. Highly efficient codecs can be obtained by ordering the operations in the enumeration. For example we may assign index 1 to the path of length $n-1$, and we may enumerate all other parent-pointer trees whose roots have indegree one before any tree whose root has indegree larger than one. The highest index may be assigned to the star graph.

Regrettably the parent-pointer data structure has essentially no error-correcting properties. An adversary who adds a single node or an edge to a parent-pointer graph g may radically change the watermark value $d(g)$ it represents.

4.2 Planted Planar Cubic Trees

Stronger error-correcting properties can be obtained by restricting the class of trees, for an adversary's modifications may result in a watermark that is outside our class, and our decoder may have a reasonably-efficient algorithm for correcting the "error" introduced by the adversary.

For example, we may choose our watermark from the class $\mathbb{G}_{PPCT}$ of planted planar cubic trees; see Fig. 3(e). The enumeration of these trees is especially straightforward and efficient, for they follow a Catalan recurrence [9]. This class has moderate data rate, approximately 0.5 bits per word. The class $\mathbb{G}_{PPCT}$ is very stealthy, for it may be represented by the commonly-encountered binary tree data structure, although it has two quirks. Each non-leaf node has exactly two children, except for a root node that has outdegree one. The underlying undirected graph is thus cubic (uniform degree 3) except at its leaves and its root. Such data structures are used in any program involving binary search trees, and superficially similar structures arise in any program that has a datatype with exactly two pointer fields.

When we represent a PPCT in a data structure, we might distinguish leaf nodes from non-leaf nodes by using null pointers in their "lchild" and "rchild" fields. However, with this representation, node-addition attacks will be potent (so long as the adversary knows enough not to create nodes with out-degree 1). Edge-addition attacks are also potent, for the decoder will be faced with a combinatorial explosion when trying to decide "which one of the three outgoing edges from a node is bogus?".

The error-correcting properties of PPCTs can be greatly improved, at a modest cost in stealthiness, by using an alternative distinction between leaf and non-leaf nodes. Leaf nodes should have a self-reference in their "rchild" field; non-leaf nodes never refer to themselves. The "lchild" fields of leaf nodes should form a singly-linked list of all the leaf nodes in reverse depth-first search order. The root node of a PPCT should be lchild-linked from the first leaf of the tree, and it should lchild-point to the last leaf of the tree; see Fig. 3(e). We have thus constructed a biconnected digraph as a supergraph of our (child-directed) binary tree, by drawing a directed outercycle linking the root and all the leaves on a planar map of the tree. Including an outercycle in the PPCT representation increases their error-correcting properties. In particular, PPCTs can detect and correct one occurrence of node deletion or insertion. It is also possible to repair edge-flips but we leave the details out of this extended abstract.

Multiple edge- and node-deletions are more problematic, for these may disconnect the PPCT. The PPCT may be disconnected into k fragments by $O(k)$ deletions, and we suspect that time exponential in k may be required for the decoder to examine all "legal" reassemblies into PPCTs.

4.3 List-type Graphs

Radix-list watermark graphs $\mathbb{G}_r$ of order n have the following structure: they are a singly-linked circular list of n nodes, in which each node has an additional pointer field that may point either to NULL or to any other node in the list; see Fig. 3(b). Thus $|\mathbb{G}_r| = n^{n+1}$, and the data rate is very close to $(\lg n)/2$. When $n = 255$, these graphs have a data rate of 4 hidden bits per word of overhead [4]. They are reasonably stealthy, for some programs employ circular lists of nodes that contain a single additional pointer, and many programs use nodes that have two pointers. However these graphs have poor error-correcting properties, for they have very little redundancy other than the n-cycle. Node- and edge-addition attacks on such graphs will force our decoder to enumerate all Hamiltonian subgraphs, in order to construct all possible watermark graphs that could have existed prior to the attacks.

We can add redundancy to a radix-list watermark graphs by restricting these to indegree two and outdegree two. The first outgoing arc on each node defines a Hamiltonian cycle. The second arc defines a permutation on the nodes. We have $|\mathbb{G}_p| = n!$ for this class of "permutation watermark graphs", and its data rate is $(\lg n)/2 - O(1)$; see Fig. 3(a). Efficient codecs may be built from textbook enumerations of permutations, e.g. [16].

Permutation watermark graphs also have modest error-correcting properties, for the decoder will gain clues about bogus arcs and nodes by noting where in- and out-degrees differ from the required value 2.

4.4 Reducible Permutations

We now introduce a class of list-like watermark graphs with excellent error-correcting properties. Our reducible permutation graphs (RPG) are reducible flow graphs with Hamiltonian paths consisting of four pieces; see Fig. 3(c):

- **header node**: This is the only node in the graph with in-degree zero. It has out-degree one, and every other node in the graph is reachable from it. Any flow graph must have such a node.
- **preamble**: These nodes immediately follow the header node and have no forward edges, except those edges used in the graph's Hamiltonian path. Because of that, edges can be inserted from anywhere in the body to anywhere in this section without making the flow graph irreducible.
- **body**: Edges between these nodes encode a self-inverting permutation. The body starts with the first node along the Hamiltonian path with a forward edge that is not part of that Hamiltonian path. In many cases, the fact that the permutation is self-inverting eliminates the need for a preamble.
- **footer node**: This node has out-degree zero, and it is reachable from every other node in the graph. Any flow graph must have a node like this as well.

Much of the information needed to extract the encoded permutation is contained in the body. The Hamiltonian path, of which there can be at most one in any reducible flow graph, puts nodes in the order by which

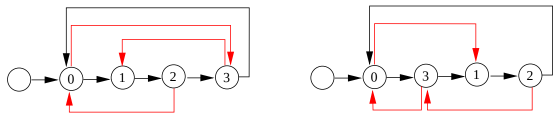


Fig. 4. Isomorphic permutation graphs encoding distinct permutations.

they have been numbered. Other edges in the body put the nodes in the order $\sigma(0), \sigma(1), \ldots, \sigma(k-1)$, where k is the number of nodes in the body and σ is the permutation being encoded.

The graph encodes a permutation that is its own inverse on the set $\{0, 1, \ldots, k-1\}$ that fixes 0. It does so with a cycle among the nodes in the body, except some edges from that cycle cannot be included in the graph because they would make the graph irreducible. Edges from nodes in the body to nodes in the preamble serve to remove the ambiguity created by these missing edges. In Fig. 3(c), nodes 5, 6, 9, and 10 were the tail nodes of such removed edges, since their out-degree to other nodes in the body is only 1. The edges from these nodes to the preamble, or lack thereof, indicates how they are to be sorted in order of the head nodes of their removed edges. No edge to the preamble means the node is to be inserted at the end of the list. An edge to the first node in the preamble, such as node 10 has, indicates that the node is to be inserted before the last node in the list. This puts the nodes in the order 5, 6, 10, 9. If nodes with removed out-edges had to be placed earlier in the list than the second to last position, we would need more nodes in the preamble. Nodes that are the head nodes of removed edges are nodes 2, 3, 4, and 7. Thus, in order to complete the cycle, we add edges from 5 to 2, from 6 to 3, from 10 to 4, and from 9 to 7. Then the nodes in permuted order are 0, 5, 2, 10, 4, 1, 9, 7, 8, 6, and 3. In cycle notation, this permutation is $(0)(1,5)(2)(3,10)(4)(6,9)(7)(8)$, which is its own inverse and is permutation number 5487 in our enumeration. We discuss the RPG encoding in the next section.

5 Properties of Reducible Permutation Encodings

For most of the encoding schemes described in the previous section, it is assumed that the order of outgoing edges is preserved. Encodings that rely on such assumptions are vulnerable to the simplest type of attacks such as edge-flips. Without this assumption, a single permutation graph can be viewed as encoding two different permutations; see Fig. 4.

The reducible permutation graphs (RPG), however, do not rely on any such assumptions. While their data rate is slightly lower than that of other encoding schemes, we believe their superior error correcting properties make up for it. Moreover, RPGs are very stealthy and fit well into the CT algorithm. We study the properties of RPGs in detail below.

Theorem 1. *A reducible permutation graph $G = (V, E)$ can be created in polynomial time.*

Proof: Let σ be a permutation on $\{0, 1, \ldots, k-1\}$ (which will henceforth be referred to as $\mathbb{Z}_k$) such that $\sigma(0) = 0$ and $\sigma = \sigma^{-1}$. We use the following algorithm to create a reducible permutation graph $G = (V, E)$:

1. Initialization: $V := \{h, f, v_0, v_1, \ldots, v_{k-1}\}$ and $E := \{(v_0, v_1), (v_1, v_2), \ldots, (v_{k-2}, v_{k-1}), (v_{k-1}, f)\}$, where h is the header, f the footer, and the remaining nodes are the body.
2. For each $i \in \mathbb{Z}_k \backslash \{0\}$ such that $\sigma(i-1) < \sigma(i)$, set $E := E \cup \{(v_{\sigma(i-1)}, v_{\sigma(i)})\}$. At this point G is still acyclic and hence reducible.
3. Compute the dominators of $G - \{h\}$, using v_0 as the root.
4. Initialize $B := \{j \in \{0, 1, \ldots, k-2\} : \sigma(j) > \sigma(j+1)\}$. Initialize $L := \emptyset$.
5. For each $j \in B$, if $v_{\sigma(j+1)}$ dominates $v_{\sigma(j)}$, set $E := E \cup \{(v_{\sigma(j)}, v_{\sigma(j+1)})\}$; else set $L := L \cup \{(v_{\sigma(j)}, v_{\sigma(j+1)})\}$. In the cases where an edge is added to E, it is a backedge, and hence has no effect on the dominance hierarchy.
6. Let $\tau : L \to \mathbb{Z}$ be defined as $\tau((v_i, v_j)) = |\{(v_{i'}, v_{j'}) \in L : i' < i, j' > j\}|$. Let $m = \max \tau(L)$.
7. If $m = 0$, set $E := E \cup \{(h, v_0)\}$ and return $G = (V, E)$.

8. Else, set $V := V \cup \{p_1, p_2, \ldots, p_m\}$. For each $(v_i, v_j) \in L$, if $\tau((v_i, v_j)) > 0$, set $E := E \cup \{(v_i, p_{\tau((v_i, v_j))})\}$, where $\{p_1, p_2, \ldots, p_m\}$ are the preamble vertices. Set $E := E \cup \{(h, p_m), (p_m, p_{m-1}), \ldots, (p_2, p_1), (p_1, v_0)\}$ and return $G = (V, E)$.

The output graph is reducible up to and including step 7. If step 8 is reached (the graph needs a preamble), the preamble is inserted so that the only path from h to any of $\{v_0, v_1, \ldots, v_{k-1}, f\}$ passes through v_0. As a result, dominance relations among these vertices are preserved, and the whole preamble dominates the whole body and the footer node. □

Theorem 2. *Any acyclic digraph has at most one Hamiltonian path.*

Proof: Let G be an acyclic digraph. Let $P_0 = v_0 v_1 \cdots v_{n-1}$ be a Hamiltonian path in G. Assume there exists some other Hamiltonian path $P_1 = v_{i_0} v_{i_1} \cdots v_{i_{n-1}}$ in G. Then there exist j and k such that $j < k$ and $i_j > i_k$. There is a path from v_{i_j} to v_{i_k} in path P_1 and a path from v_{i_k} to v_{i_j} in path P_0. This contradicts our assumption that G is acyclic, so G must have at most one Hamiltonian path. □

Theorem 3. *Any reducible flowgraph has at most one Hamiltonian path. Furthermore, this Hamiltonian path can be found in polynomial time if it exists.*

Proof: Let G be a reducible flowgraph, and let vv' be a backedge in G. Then v' dominates v. By definition, any path to v from the root of G passes through v'. Since any Hamiltonian path in G must begin at its root, the edge vv' cannot be contained in any Hamiltonian path of G, because v' will have already been used by the time the path reaches v. So, any Hamiltonian path in G is contained in the acyclic subgraph of G. By the above theorem G has at most one Hamiltonian path. The path can be found in polynomial time as follows:

1. For each edge uv in $G = (V, E)$, assign that edge a weight of ∞ if v dominates u and a weight of -1 otherwise. Then the subgraph of G with finite edge weights is acyclic.
2. Run the Bellman-Ford shortest path algorithm on G. G has a Hamiltonian path iff the weight of that path is $-|V| + 1$; see [12] for more details.

□

Theorem 4. *Reducible permutation graphs can be decoded and corrected for edge-flips in polynomial time.*

Proof: The following algorithm decodes an RPG $G = (V, E)$:

1. Compute the dominance tree of G and verify that G is reducible.
2. Find a Hamiltonian path P in G (in polynomial time from above theorem).
3. Let v_0 be the first node with a forward edge not in P. Let $v_1, v_2, \ldots, v_{k-1}$ be the subsequent nodes in P, excluding f. Let $p_m, p_{m-1}, \ldots, p_1$, be the nodes strictly between h and v_0, if any.
4. Let $B := \{v_0, v_1, \ldots, v_{k-1}\}$.
5. Create an empty ordered list of nodes L. For each $i \in \{0, 1, \ldots, k-1\}$, if v_i has one outgoing edge to $B \cup \{f\}$, insert v_i in L. If v_i has no outgoing edges to $\{p_1, p_2, \ldots, p_m\}$, insert it at the end of the list. Otherwise, if it has an edge to p_j, insert it before the last j elements of the list.
6. For each $i \in \{0, 1, \ldots, k-1\}$, if v_i has only one incoming edge from $B \cup \{h, p_1\}$, add an edge from the first element of L to v_i and remove that first element from L.
7. Initialize an integer array $A[]$ of k elements. Set $A[0] := 0$.
8. For each $i \in \{1, 2, \ldots, k-1\}$, there will now be exactly one edge from $v_{A[i-1]}$ to $B \cup \{f\}$ not contained in the path P. If that edge goes to a vertex v_j, set $A[i] := j$. If that edge goes to f, set $A[i] := A[i-1] + 1$.
9. A will now contain the appropriate permutation.

Since the order of the vertices is determined by the Hamiltonian path P, and since any reducible flow graph has at most one Hamiltonian path, this algorithm will extract the same permutation from any two isomorphic graphs. Therefore, the RPG encoding scheme is resilient to edge flipping attacks. □

We have shown how to encode and decode RPGs. Now we will place a lower bound on how much data an RPG can encode per node. To do this, we use the following lemma, which places a lower bound on the number of self-inverting permutations on n elements. Self-inversion of the RPGs is useful as it allows us (most of the time) to decode the graph without the use of any preamble. The preamble is not inserted if it is not necessary and thus the size of the RPGs is smaller on average.

Theorem 5. *The data rate for RPGs is at least* $(\lg n - \lg e - 1)/4$.

Proof: See Appendix. □

6 Summary

We have presented several classes of graphs that can be used by two graph-based algorithms for software watermarking and fingerprinting. We have also formulated an attack model, considering (by order of difficulty) the types of changes that an adversary can make to a watermarked program. We have studied the error-correcting properties of the classes of graphs discussed in the paper and have considered the trade-offs between resiliency, data rate, performance, and stealthiness. While we have yet to find an excellent encoding scheme, we believe that the reducible permutations graphs (RPGs) and planted planar cubic trees (PPCTs) offer the greatest potential.

A number of software watermarking algorithms (including all the encoding schemes discussed in this paper) have been implemented within the SandMark software protection research tool. It can be downloaded from `http://sandmark.cs.arizona.edu`.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.
2. R. J. Anderson and F. A. Peticolas. On the limits of steganography. *IEEE J-SAC*, 16(4), May 1998.
3. W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM Systems Journal*, 35(3&4):313–336, 1996.
4. C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, San Antonio, TX, Jan. 1999.
5. C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages, ICCL'98*, Chicago, IL, May 1998.
6. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, Jan. 1998.
7. R. L. Davidson and N. Myhrvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884, Sept. 1996. Assignee: Microsoft Corporation.
8. R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL'96*, pages 1–15, St. Petersburg Beach, Florida, 21–24 Jan. 1996.
9. F. Harary and E. Palmer. *Graphical Enumeration.* Academic Press, New York, 1973.
10. S. Kannan and T. A. Proebsting. Register allocation in structured programs. *Journal of Algorithms*, 29(2):223–237, Nov. 1998.
11. D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, USA, third edition, 1997.
12. E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Dover Publications, Inc., 2001.
13. G. Qu and M. Potkonjak. Analysis of watermarking techniques for graph coloring problem. In *IEEE/ACM International Conference on Computer Aided Design*, pages 190–193, Nov. 1998.
14. G. Ramalingam. The undecidability of aliasing. *ACM TOPLAS*, 16(5):1467–1471, Sept. 1994.
15. J. P. Stern, G. Hachez, F. Koeune, and J.-J. Quisquater. Robust object watermarking: Application to code. In *Information Hiding*, pages 368–378, 1999.
16. A. Tucker. *Applied Combinatorics.* Wiley, 3rd edition, 1994.
17. S. A. Vanstone and P. C. V. Oorschot. *An Introduction to Error Correcting Codes with Applications.* Klewer Academic Publishers, 1989.
18. R. Venkatesan, V. Vazirani, and S. Sinha. A graph theoretic approach to software watermarking. In *4th International Information Hiding Workshop*, Pittsburgh, PA, Apr. 2001.

Appendix

Theorem 5. *The data rate for RPGs is at least* $(\lg n - \lg e - 1)/4$.

Proof: Let $p : \{0,1,2,\ldots,n\} \to \{0,1,2,\ldots,n\}$ be a permutation such that $p(0) = 0$. We first show that there exist permutations $s : \{0,1,2,\ldots,n\} \to \{0,1,2,\ldots,n\}$ and $t : \{0,1,2,\ldots,n\} \to \{0,1,2,\ldots,n\}$ such that $s(0) = t(0) = 0$, $s = s^{-1}$, $t = t^{-1}$, and $p = t \circ s$.

Let $p = p_r \circ p_{r-1} \circ \cdots \circ p_1$, where p_1 through p_r are disjoint cycles. For each $k \in \{1,2,\ldots,r\}$, define $S_k = \{i \in \{0,1,2,\ldots,n\} \mid p_k(i) \neq i\}$. Note that $0 \notin \bigcup_{k=1}^{r} S_k$.

Let $k \in \{1,2,\ldots,r\}$. Let $p_k = (a_1 a_2 \cdots a_m)$. That is, $p_k(a_i) = a_{i+1}$ for each $i \in \{1,2,\ldots,m-1\}$, $p_k(a_m) = a_1$, and $S_k = \{a_1,a_2,\ldots,a_m\}$. We will define permutations $s_k : \{0,1,2,\ldots,n\} \to \{0,1,2,\ldots,n\}$ and $t_k : \{0,1,2,\ldots,n\} \to \{0,1,2,\ldots,n\}$ such that $p_k = t_k \circ s_k$, $s_k = s_k^{-1}$, $t_k = t_k^{-1}$, and $s_k(i) = t_k(i) = i$ for all $i \notin S_k$. Let

$$t_k(j) = \begin{cases} a_{m+2-i}, & \text{if } j = a_i \text{ for some } i > 1; \\ j, & \text{otherwise,} \end{cases}$$

and let

$$s_k(j) = \begin{cases} a_{m+1-i}, & \text{if } j = a_i \text{ for some } i; \\ j, & \text{otherwise.} \end{cases}$$

If $j \notin S_k$, $t_k(s_k(j)) = t_k(j) = j$. $t_k(s_k(a_m)) = t_k(a_1) = a_1$. If $i < m$, $m+1-i > 1$, so $t_k(s_k(a_i)) = t_k(a_{m+1-i}) = a_{m+2-(m+1-i)} = a_{i+1}$. So, $t_k \circ s_k = p_k$.

If $j \notin S_k$, $s_k(s_k(j)) = s_k(j) = j$. For any $a_i \in S_k$, $s_k(s_k(a_i)) = s_k(a_{m+1-i}) = a_i$. So, s_k is a permutation and $s_k = s_k^{-1}$.

If $j \notin S_k$ or $j = a_1$, $t_k(t_k(j)) = t_k(j) = j$. For any $i > 1$, $m+2-i > 1$, so $t_k(t_k(a_i)) = t_k(m+2-i) = a_i$. So, t_k is a permutation and $t_k = t_k^{-1}$.

In this manner we can construct an s_k and a t_k for each p_k.

Let $j \neq k$. If $i \in S_j$, $i \notin S_k$ and $t_j(i) \notin S_k$, so $t_j(s_k(i)) = t_j(i) = s_k(t_j(i))$. Similarly, $t_j(s_k(i)) = s_k(t_j(i))$ if $i \in S_k$ or if i is in neither S_j nor S_k. So, $t_j \circ s_k = s_k \circ t_j$ for any $j \neq k$.

So, letting $s = s_r \circ s_{r-1} \circ \cdots \circ s_1$ and $t = t_r \circ t_{r-1} \circ \cdots \circ t_1$, we have that

$$\begin{aligned} p &= p_r \circ p_{r-1} \circ \cdots \circ p_1 \\ &= t_r \circ s_r \circ t_{r-1} \circ s_{r-1} \circ \cdots \circ t_1 \circ s_1 \\ &= t_r \circ t_{r-1} \circ \cdots \circ t_1 \circ s_r \circ s_{r-1} \circ \cdots \circ s_1 \\ &= t \circ s. \end{aligned}$$

If $i \in S_k$, then $s(s(i)) = s_k(s_k(i)) = i$ and $t(t(i)) = t_k(t_k(i)) = i$. If $i \notin \bigcup_{k=1}^{r} S_k$, then $s_k(i) = t_k(i) = i$ for all k, so $s(i) = t(i) = i$. So, $s = s^{-1}$ and $t = t^{-1}$. Since $0 \notin \bigcup_{k=1}^{r} S_k$, $s(0) = t(0) = 0$.

If I_n is the set of self-inverting permutations on n elements, we have a surjection from $I_n \times I_n$ to S_n. Thus, $|I_n| \geq \sqrt{n!}$. So, the number of bits encoded by a reducible permutation graph with a k-node body is $\lfloor \lg |I_n| \rfloor \geq \left\lfloor \frac{1}{2} \lg n! \right\rfloor$. As long as $n \geq 2$, such as graph has at most $2n$ nodes, so the bitrate is at least

$$\frac{\left\lfloor \frac{1}{2} \lg n! \right\rfloor}{2n} \geq \frac{n \lg n - n \lg e + \lg e - 2}{4n} = \frac{1}{4} \lg n - \frac{1}{4} \lg e + \frac{\lg e}{4n} - \frac{1}{2n} \geq \frac{1}{4} (\lg n - \lg e - 1)$$

bits per node.

□